Algoritmi e Strutture Dati – 20/06/2023

Esercizio A1 – Complessità – Punti ≥ 8

Dal codice, è possibile vedere che la matrice di input è quadrata. Consideriamo, per semplicità, il caso in cui n è una potenza di 2. Il risultato è valido anche per dimensioni che non rispettano tale vincolo.
Il codice calcola la somma dei valori lungo i bordi della matrice, in tempo $O(n)$, e poi chiama ricorsivamente se stessa su due porzioni di dimensione $n/2$, che corrispondono al secondo e quarto quadrante se interpretiamo la matrice come un piano cartesiano.
La funzione di complessità è quindi:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

Applicando il Master Theorem, otteniamo $\alpha = \log_2 2 = 1$ e $\beta = 1$. Poichè $\beta = \alpha$, per il Master Theorem $T(n) = \Theta(n \log n)$.

Esercizio A2 – Somma su albero – Punti ≥ 10

Il problema può essere risolto con una semplice visita in profondità. Assumiamo che ∞ sia un valore speciale che non compare in nessun nodo. Il costo computazionale è ovviamente quello di una visita di un albero, cioè $O(n)$.

```
boolean isSumTree(Tree T)
    int sum = isRec(T)
    return sum ≠ ∞ and T.value == sum
```

```
int isRec(Tree T)
    if T == nil then
    |   return 0
    else if T.left() == nil and T.right() == nil then
    |   return T.value
    int sum_L = isRec(T.left())
    int sum_R = isRec(T.right())
    if sum_L ≠ ∞ and sum ≠ ∞ and T.value == sum_L + sum_R then
    |   return T.value · 2
    else
    └   return ∞
```

Esercizio A3 – Tree fire – Punti ≥ 12

L'algoritmo proposto effettua una visita in profondità dell'albero come se fosse un grafo, a partire dal nodo t. Invece di memorizzare informazioni sulla visita sotto forma di valori visited (che dovrebbero essere inseriti all'interno dell'albero oppure in qualche forma di tabella hash), l'algoritmo semplicemente evitare di tornare sui propri passi, sfruttando il fatto che non esistono cicli. Viene quindi passato il riferimento *prev* al nodo precedente, che serve ad evitare di tornare indietro.
Il costo computuazionale è quello di una visita in profondità, cioè $O(n)$.

```
int fire(Tree t)
    return fireRec(t, nil)
```

```
int fireRec(Tree t, Tree prev)
    int max = 0
    if t.left ≠ nil and t.left ≠ prev then
    |   max = max(max, 1 + fireRec(t.left, t))
    if t.right ≠ nil and t.right ≠ prev then
    |   max = max(max, 1 + fireRec(t.right, t))
    if t.parent ≠ nil and t.parent ≠ prev then
    |   max = max(max, 1 + fireRec(t.parent, t))
    return max
```

Esercizio B1 – Misciotto 1 – Punti ≥ 8

Dovendo stampare un insieme combinatorio di stringhe, utilizziamo un algoritmo di backtracking. La funzione printMixing() è un wrapper che richiama printRec, dopo aver creato un vettore T contenente spazio per $n_1 + n_2$ caratteri.
La funzione ricorsiva lavora dal fondo, per passare meno parametri. Quando entrambi n_1 e n_2 sono a zero, la stringa viene stampata; altrimenti, la funzione viene chiamata ricorsivamente, prima aggiungendo un carattere della prima stringa (se possibile), poi della seconda (se possibile).

```
printMixing(Item[] S1, Item[] S2, int n1, intn2)
    int[] T = new int[1...n1 + n2]
    printRec(S1, S2, T, n1, n2)
```

```
printRec(Item[] S1, Item[] S2, Item[] T, int n1, int n2)
    if n1 == 0 and n2 == 0 then
        print T
    else
        if n1 > 0 then
            T[n1 + n2] = S1[n1]
            printRec(S1, S2, T, n1 − 1, n2)
        if n2 > 0 then
            T[n1 + n2] = S2[n2]
            printRec(S1, S2, T, n1, n2 − 1)
```

Viste le due chiamate ricorsive, possiamo dire che il numero di stringhe stampate è limitato superiormente da $2^{n_1+n_2}$; poichè la stampa ha costo $O(n_1 + n_2)$, l'algoritmo ha costo totale $O((n_1 + n_2) \cdot 2^{n_1+n_2})$.
Questo limite superiore però non è stretto. È possibile calcolare il numero di stringhe da stampare, pari a

$$\frac{(n_1 + n_2)!}{n_1! \cdot n_2!}$$

Questo corrisponde alle permutazioni di $n_1 + n_2$ caratteri, tenuto conto però che prendiamo una sola delle n_1 permutazioni dei caratteri di S_1 e una sola delle n_2 permutazioni dei caratteri di S_2.
Il costo corretto è quindi

$$T(n) = O\left((n_1 + n_2) \cdot \frac{(n_1 + n_2)!}{n_1! \cdot n_2!}\right)$$

Esercizio B2 – Misciotto 2 – Punti ≥ 12

Innanzitutto, perché ormai perdo colpi, il problema era già stato proposto nel gennaio 2016 e non me ne sono reso conto. Per la soluzione, fate quindi riferimento a quel compito. Qui, ci concentriamo su un paio di soluzioni errate che sono state proposte da molti.

Soluzione basata su LCS

Alcuni hanno calcolato utilizzato la condizione $\mathsf{LCS}(S_1, T) + \mathsf{LCS}(S_2, T) = n$. Questo algoritmo non funziona, perché $\mathsf{LCS}(\text{"ABC"}, \text{"ABCDEF"}) + \mathsf{LCS}(\text{"ABC"}, \text{"ABCDEF"}) = 6$ ma `"ABCDEF"` non è un misciotto di `"ABC"` e `"ABC"`.

Soluzione lineare

L'idea è la seguente: si confrontano i caratteri della stringa T con i caratteri delle stringhe S_1 e S_2, in ordine. Se si trova una corrispondenza, si avanza nella stringa in cui c'è un carattere uguale; se non si trova una corrispondenza, si ritorna **false** per indicare che T non è un misciotto di S_1 e S_2.

```
isMixing(Item[] S1, Item[]S2, Item[] T, int n1, int n2, int n)
if n1 + n2 ≠ n then
    return false
else
    int i1 = 1
    int i2 = 1
    for i = 1 to n do
        if i1 ≤ n1 and S1[i1] == T[i] then
            i1 = i1 + 1
        else if i2 ≤ n2 and S2[i2] == T[i] then
            i2 = i2 + 1
        else
            return false
    return true
```

Il costo di questo algoritmo è $O(n)$, ma purtroppo è errato. Si considerino per esempio le stringhe S_1 = `"AC"`, S_2 = `"AB"`, T = `"ABAC"`.
L'algoritmo incontra la prima lettera A e avanza su S_1; a questo punto, il prossimo carattere di T è `"B"`, ma i prossimi caratteri di S_1 e S_2 sono `"C"` e `"A"`, rispettivamente. L'algoritmo quindi ritorna **false**, ma ovviamente `"ABAC"` è un misciotto di `"AC"` e `"AB"`.
Provare entrambe le strade richiede un algoritmo ricorsivo, che può dare origine a un costo esponenziale quando le stringhe sono composte da un unico carattere ripetuto n volte; per questo motivo, utilizziamo la programmazione dinamica per evitare di ricalcolare casi già calcolati.

Esercizio B3 – Conferenza – Punti ≥ 10

Il problema può essere risolto tramite una rete di flusso. La prima cosa da osservare è che non sono necessari nodi per le aree scientifiche, i quali servono solo per associare reviewer e paper. Esaminiamo innanzitutto una soluzione non corretta, proposta da molti, che consente a un reviewer di effettuare più di una review per lo stesso paper.

Soluzione errata

Creiamo i seguenti nodi:

- una sorgente s;
- un nodo r_i per ogni reviewer (n_r nodi);
- due nodi per ogni paper p_j, etichettati p_j^{ex} (EXpert) e p_j^{ne} (Non Expert); p_j^{ex} serve a raccogliere i due reviewer esperti, p_j^{ne} serve a raccogliere il terzo reviewer (esperto oppure no) ($2n_p$ nodi);
- un pozzo t.

Creiamo i seguenti archi:

- colleghiamo s a ogni reviewer, con peso k per limitare il numero di review (n_r archi);
- colleghiamo ogni reviewer r_i ad uno nodo p_j^{ex} se e solo $a_j \in E_{r_i}$, cioè se il reviewer r_i è esperto nell'area a_j del paper p_j, con peso 1 ($n_r \cdot n_p$ archi o meno);
- colleghiamo ogni reviewer r_i a tutti i nodi p_j^{ne} ($n_r \cdot n_p$ archi);
- colleghiamo tutti i nodi p_j^{ex} e p_j^{ne} al pozzo ($2n_p$ archi), con peso 2 e 1, rispettivamente.

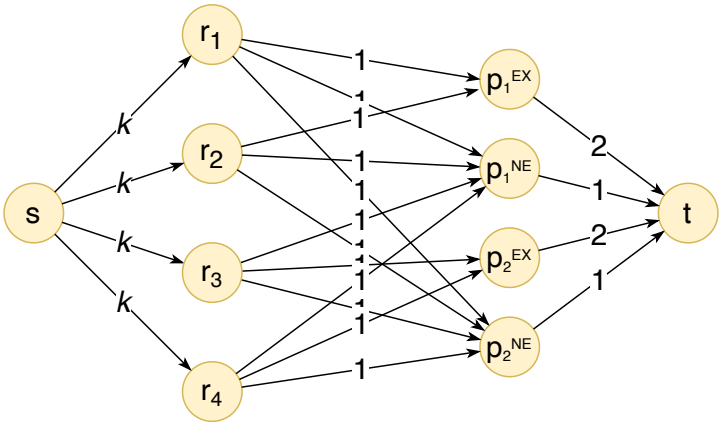
La dimensione di V ed E è la seguente:

$$|V| = 2 + n_r + 2n_p$$
$$|E| \leq n_r + 2n_p + 2n_r n_p$$

Utilizzando il limite di Ford-Fulkerson e tenuto conto che il flusso massimo è limitato superiormente da $3n_p$, il costo totale dell'algoritmo proposto è

$$3n_p \cdot (|V| + |E|) \leq 3n_p(2 + 2n_r + 4n_p + 2n_r n_p) = O(n_p^2 n_r)$$

La figura seguente mostra un piccolo esempio con 3 reviewer e due paper.



Il problema di questa soluzione è che un reviewer può contribuire due volte per ogni paper, una volta come esperto e una volta come non-esperto, e questo non è accettabile. Il motivo è che abbiamo creato un arco fra un reviewer ed entrambi i nodi associati al paper.

Soluzione corretta – Versione 1

È possibile modificare questa situazione creando un solo arco da ogni reviewer alla coppia di nodi associati al paper p_i - o verso il nodo p_j^{ex}, se il reviewer è effettivamente esperto, o verso il nodo p_j^{ne}. In questo modo, si evita la doppia review.
Tuttavia, questo approccio porta ad un altro problema: diventa obbligatorio che un paper abbia due reviewer esperti e uno non esperto. Ma potrebbe accadere che tutti i reviewer siano esperti per un particolare paper, e quindi nessuno potrebbe assumere il ruolo di reviewer non esperto.
Un'alternativa sarebbe aumentare a 3 la capacità dell'arco dal nodo p_j^{ex} al pozzo, ma questo significherebbe che potrebbero esserci fino a quattro review per paper, il che è contrario alle specifiche.
Dunque, aggiungiamo un altro nodo, il cui compito è ridurre il numero effettivo di review a 3.
Creiamo i seguenti nodi:

- una sorgente s;
- un nodo r_i per ogni reviewer (n_r nodi);
- tre nodi per ogni paper p_j, chiamati p_j^{ne}, p_j^{ex} e p_j^{tot}; p_j^{ne} raccoglie i reviewer non esperti, p_j^{ex} raccoglie i reviewer esperti mentre p_j^{tot} serve a raccogliere tre reviewer in totale ($3n_p$ nodi);
- un pozzo t.

Creiamo i seguenti archi:

- colleghiamo s a ogni reviewer, con peso k (in quanto può fare al massimo k review) (n_r archi);
- colleghiamo ogni reviewer r_i al nodo p_j^{ne}, se r_i non è esperto nell'area del paper p_j, con peso 1. Altrimenti, al nodo p_j^{ex} con peso 1, se r_i è esperto nell'area del paper p_j ($n_r n_p$ archi);
- colleghiamo p_j^{ne} e p_j^{ex} a p_j^{tot}, con peso 1 e 3 rispettivamente (n_p archi);
- colleghiamo tutti i nodi p_j^{tot} al pozzo con peso 3 (n_p archi).

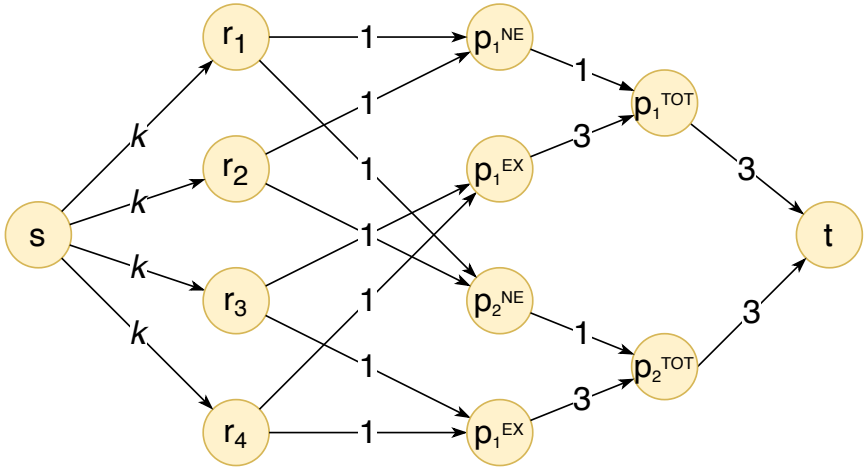
La dimensione di V ed E è la seguente:

$$|V| = 2 + n_r + 3n_p$$
$$|E| = n_r + 2n_p + n_r n_p$$

Utilizzando il limite di Ford-Fulkerson e tenuto conto che il flusso massimo è limitato superiormente da $3n_p$, il costo totale dell'algoritmo proposto è

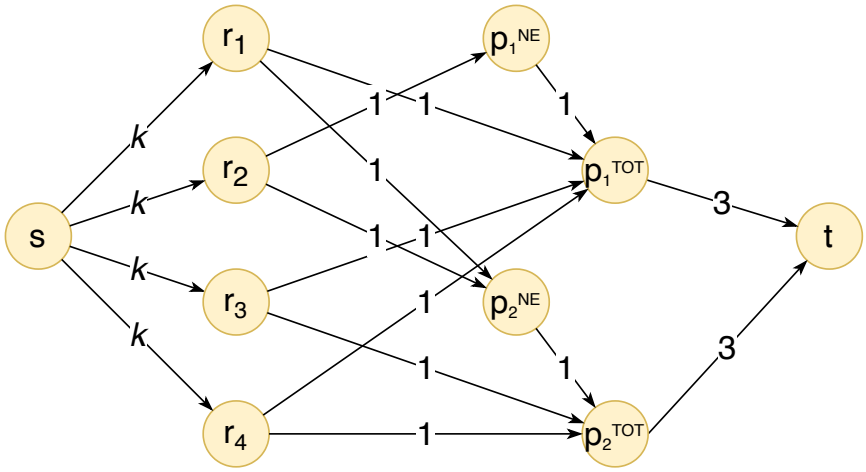
$$3n_p \cdot (|V| + |E|) = 3n_p(2 + 2n_r + 5n_p + n_r n_p) = O(n_p^2 n_r)$$

In questa configurazione, ogni reviewer ha al massimo un arco uscente per ogni paper, con peso 1; quindi contribuirà al massimo una review per quel paper.
Se il flusso è esattamente uguale a $3n_p$, ogni paper ha ricevuto esattamente tre reviewer.
La figura seguente mostra un piccolo esempio con 3 reviewer e due paper.



Soluzione corretta – Versione 1.1

È possibile ridurre leggermente il numero di nodi eliminando il nodo p_j^{ex}, facendo confluire gli archi destinato ad esso direttamente su p_j^{tot}, che "selezionerà" tre reviewer esperti o due esperti e un non esperto, preso da p_j^{ne}.
La figura seguente mostra un piccolo esempio con 3 reviewer e due paper.



La complessità resta uguale, cala leggermente il fattore moltiplicativo.

Soluzione corretta – Versione 2

Un'ulteriore soluzione – a dire il vero la prima che mi era venuta in mente – richiede di inserire, fra reviewer e paper, un nodo per ogni coppia (reviewer,paper), dal quale entra e esce un solo arco che va verso la configurazione vista sopra. Aggiungere un nodo con un solo arco entrante e un solo arco uscente equivale a inserire un arco diretto–quindi il nodo può essere eliminato. Mi capita spesso di farlo–poi quando lo disegni ti rendi conto che è inutile. In ogni caso, la complessità di questa soluzione è la stessa di quelle precedenti - aumentano solo i fattori moltiplicativi.